VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

Daniel Komínek

Bakalářská práce

Vedoucí práce: Ing. Michal Vašinek, Ph.D.

Ostrava, 2021

Abstrakt

Cílem bakalářské práce je shrnout působení ve firmě Liberty Ostrava a.s. Nejprve se pracovalo na programu PCChecker, který slouží pro práci se soubory a kontrolu disků na serveru. Část bakalářské práce je věnována projektu Technologická a finanční kalkulace vsázky pro výrobu aglomerátu a surového železa, kde bylo za úkol algoritmus kalkulace co nejvíce zrychlit. K dosažení nejlepších výsledků na výpočty pomocí grafických karet byla využita technologie OpenCL.

Klíčová slova

OpenCL, GPU, CPU, materiál, kombinace

Abstract

The aim of the bachelor thesis is to summarize the activities and tasks in the company Liberty Ostrava a.s. First, we worked on the PCChecker program, which is used to work with files and disk checking on the server. The rest of the bachelor's practice was devoted to the project Technological and financial calculation of the charge for the production of agglomerate and pig iron, where the task was to speed up the algorithm calculation as much as possible. To achieve the best results, the OpenCL technology was used for intensive calculations by using graphics cards.

Keywords

OpenCL, GPU, CPU, material, combinations

Poděkování

Děkuji vedoucímu bakalářské práce panu Ing. Michalu Vašinkovi, Ph.D. a panu Ing. Daliboru Stehlíkovi, který byl mým konzultantem za cenné rady při vypracování této bakalářské práce.

Obsah

Seznam použitých zkratek a symbolů

APU	–	Accelerated processing unit - čip, do kterého je integrován CPU a zároveň GPU
atd.	–	a tak dále
CIL	–	Common Intermediate Language
CPU	–	Central processing unit - procesor počítače
device	–	jedná se o kód spuštěný jakýmkoliv zařízením
GPU	–	Graphics processing unit - grafická karta počítače
host	–	jedná se o kód spuštěný procesorem
LINQ	–	Language Integrated Query
MB	–	megabyte
mil.	–	milión
mld.	–	miliarda
ms	–	milisekunda
např.	–	například
OpenCL	–	Open Computing Language
OS	–	Operační systém
PC	–	Personal computer - počítač
PCIe	–	Peripheral Component Interconnect express - slot, který využívá grafická karta pro propojení se základní deskou
RAM	–	Random Access Memory - operační paměť
SQL	–	Structured Query Language
tzv.	–	takzvaný, takzvaně
vs	–	versus - proti

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

V této bakalářské práci je popsáno mé působení ve firmě Liberty Ostrava a.s. Ve firmě již pracuji druhým rokem na pozici programátor-analytik, programátor MES. Prostředí i pracovní tým mi vyhovují, proto jsem se rozhodl pro vypracování bakalářské práce formou odborné praxe. Zabýval jsem se především testováním technologie OpenCL, kterou jsem využil pro výpočty na grafických kartách. Stejnou technologii jsem také využil na optimalizaci algoritmu v hlavním projektu. Většina firem dnes očekává od svých budoucích zaměstnanců několikaletou praxi v daném oboru. Při studiu jsem získal pouze základní znalosti, které jsem ve firmě a v rámci odborné praxe o hodně rozšířil.

Ve druhé kapitole jsou popsány základní informace o firmě včetně harmonogramu samotné praxe. V další kapitole jsem popsal využívané technologie pro vypracování všech zadaných úloh. Ke každé technologii jsou uvedeny základní informace, popřípadě nějaké kódy, aby čtenář pochopil, jak technologie funguje. Ve čtvrté kapitole jsou podrobně popsány všechny úlohy, které mi byly během odborné praxe přiděleny. Jednotlivé úlohy a podúlohy byly konzultovány s p. Michalem Rudinským, který se mnou pracuje v týmu a s nadřízeným p. Daliborem Stehlíkem. Všechny úlohy a podúlohy byly ve firmě nutné ke konečnému vypracování projektů. V páté kapitole popisuji znalosti a dovednosti, které jsem získal při studiu a uplatnil v průběhu odborné praxe. V poslední kapitole hodnotím celkový průběh a formu odborné praxe.

Kapitola 2

Informace o praxi

V této kapitole jsou popsány informace ohledně firmy a harmonogram absolvované praxe.

2.1 O firmě

Liberty Ostrava a.s. je integrovaný hutní podnik, který vyrobí 3,6 mil. tun oceli ročně. Využívá se hlavně ve strojírenství, stavebnictví nebo petrochemickém průmyslu. V České republice je to největší podnik na výrobu svodidel a trubek. Své výrobky dodává do 40 zemí světa. Celkově má ve všech závodech a dceřiných společnostech přibližně 6000 zaměstnanců. [1]

Díky nadstandardní ekologizaci vyrábí své výrobky s minimálním možným dopadem na životní prostředí. Huť patří do skupiny LIBERTY Steel Group, která je součástí globálního uskupení GFG Alliance, která patří rodině Sanjeeva Gupty. GFG Alliance má tři nezávislé průmyslové divize: ocelářskou LIBERTY Steel Group, hliníkářskou ALVANCE a energetickou SIMEC. Sídlí v Londýně a působí v 10 zemích světa, kde má celkem 35 000 zaměstnanců a obrat ve výši 20 mld. amerických dolarů. [1]

V roce 1942 vznikla největší hutní společnost České republiky a nesla název Vítkovické železárny. O devět let později se jmenovala Nová Huť Klementa Gottwalda. V roce 1989 se podnik stal státním a nesl název Nová Huť. [2] Roku 2003 koupil Novou Huť Lakshmi Mittal, který je majitelem firmy ArcelorMittal, největší ocelářské skupiny na světě. [3] Do července roku 2019 podnik nesl jméno ArcelorMittal Ostrava. Tohoto data se společnost stala součástí skupiny Liberty a její název se změnil na Liberty Ostrava. [2]

2.2 Harmonogram praxe

- 10.9. 2020
  Zařazení do pracovního prostředí na závodě číslo 12 - Vysoké pece.

- 15.9. 2020 - 7.10. 2020
  Práce na programu PCChecker.
- 19.10. 2020
  Telekonference s konzultantem a seznámení s hlavním projektem.
- 20.10. 2020
  Analýza projektu ve firmě formou porady.
- 22.10 2020 - 20.11. 2020
  Testovací programy a orientační časy výpočtů, program GpuCount a jeho funkce, možnosti využití GPU s jazykem C#.
- 24.11. 2020
  Algoritmus na generaci všech kombinací materiálů.
- 25.11 2020 - 26.11. 2020
  Zjišťování, jak se uvolňují jádra a vlákna při více spuštěných instancí programu.
- 30.11. 2020
  Telekonference s Michalem Rudinským, členem týmu, ohledně přepisu algoritmu z CPU na GPU.
- 7.1. 2021 - 16.2. 2021
  Přepis algoritmu z CPU na GPU.
- 17.2. 2021
  Telekonference s Michalem Rudinským ohledně mých dosavadních výsledků a časů.
- 18.2. 2021 - 16. 3. 2021
  Optimalizace algoritmu.
- 16.3. 2021
  Setkání s týmem ohledně více GPU a zkoušení a testy výpočtů s více GPU.
- 17.3. 2021 - 31. 3. 2021
  Optimalizace a úpravy algoritmu.
- 5. 4. 2021
  Návrh ideálního PC na výpočet vsázky.

Kapitola 3

Použité technologie

3.1 Programovací jazyk C#

3.1.1 Historie

CPU v počítači zvládne vykonávat jen omezené množství triviálních instrukcí. Takovou instrukcí se rozumí např. sčítaní, odčítání dvou různých adres. Instrukce jsou uloženy jako sekvence bitů. Programovací jazyky se postupně vyvíjely celkově ve třech generacích. [4]

V první generaci máme tzv. strojový kód, který obsahuje pouze hexadecimální čísla. Je pro člověka velmi složitě čitelný. Pro samotný PC je to nejrychlejší varianta.[4]

V druhé generaci máme jazyk Assembler, který je stejně složitý, jako strojový kód, ale upravený tak, aby jej člověk jednodušeji přečetl. [4]

Do třetí generace jazyků přišla velká změna, která byla zaměřena na ještě větší čitelnost a přehlednost zdrojového kódu. Klade se zde důraz především na to, jak program vidí lidé a ne samotný počítač. Čísla se ukládají do proměnných a samotný zdrojový kód je spíše podobný matematickému zápisu. Právě do třetí generace patří již zmíněný jazyk C#. [4]

3.1.2 .NET framework

[4] Skládá se ze čtyř následujících částí:

1. Výběr z několika jazyků - C, C++, C# atd.
2. Microsoft Visual Studio - prostředí, ve kterém se píše kód. IDE - Integrated Development Environment
3. Virtuální stroj, který interpretuje mezikód CIL do instrukcí fyzického CPU
4. Knihovny - hlavní výhoda .NETu. Je zde na výběr z mnoha knihoven a komponent, které lze v programu využívat

3.1.3 C#

Je jednoduchý, moderní, multiplatformní a objektově-orientovaný programovací jazyk, který má své kořeny v jazycích C a C++. Některé funkce převzal také z jazyku Java. Jazyk přináší mnoho změn a nových užitečných funkcí oproti C, jako např. GC (garbage collector), který se sám na pozadí stará o uvolňování již nepotřebných objektů z paměti, zpracování výjimek, poskytuje strukturovaný a rozšiřitelný přístup k detekci a obnově chyb. Je typově bezpečný, což znemožňuje např. číst z neinicializovaných proměnných, číst z polí nad rámec jejich hranic nebo pro provedení nezaškrtnutých přetypování typu. [5]

Všechny datové typy C# jako jsou např. int, float, double nebo string, dědí z kořenového typu object. Díky této vlastnosti všechny datové typy sdíli mnoho běžných operací. [5]

[6] **Výhody:**

- Je velice efektivní ve správě systému.
- Nemá problém s tzv. memory leakem.
- Náklady na údržbu jsou nižší a je bezpečnější na spuštění v porovnání s ostatními jazyky.
- Zdrojový kód je kompilovaný do standardního jazyka CIL nezávisle na cílovém OS.
- C# lze využít na vývoj desktopových aplikací, webových aplikací či webových služeb.
- Je využíván i na vývoj her v herním enginu Unity.

[6] **Nevýhody:**

- Spouští se pomaleji než např. C/C++.
- Aplikace musí být kompilována při každé změně.
- Na cílovém PC musí být instalovaná požadovaná verze .NETu, která musí být podporovaná OS počítače.

3.2 LINQ

Je knihovna příkazů, která umožňuje používání dotazů, jak známe z databází, např. jazyk SQL přímo v jazyce C#. Dotazy jsou vyhodnocovány tzv. línou formou, což znamená, že se dotaz zpracuje až ve chvíli, kdy je k jeho výsledku přistupováno. Díky této funkci zbytečně nezpomaluje a nezatěžuje PC, když to zrovna není nutné. Pomocí LINQ příkazů lze kolekci seřazovat, seskupovat nebo filtrovat podle různých požadavků s minimálním množstvím kódu. LINQ je vyhodnocován sekvenčně na jednom CPU vlákně. Paralelní alternativou je PLINQ, který lze využít přidáním funkce .AsParallel() k dotazu. [7]

3.3 SQL

Je programovací jazyk, který se využívá pro komunikaci s databází. SQL je standardní jazyk pro relační databáze a správu systémů. SQL příkazy se využívají např. na aktualizaci, přidávání a odebírání dat v databázi nebo na jejich získání. Mezi nejznámější databázové systémy patří Oracle, Microsoft SQL Server, Access atd. Každá z těchto služeb má trochu jinou syntaxi, ale klíčové příkazy jako jsou Select, Insert, Update, Delete, Create nebo Drop, jsou využívány stejně. [8]

3.4 RabbitMQ

RabbitMQ je software pro řazení zpráv. Jsou zde definovány fronty, ke kterým se aplikace připojují za účelem přenosu zpráv. Může se jednat o zprávu, která obsahuje jakoukoliv informaci, např. informace o procesu, jeho návratovém kódu či návratové chybě nebo to může být jednoduchá textová zpráva. V našem konkrétním využití se tento software využije na kolekci všech spuštěných aplikací, které tuto funkci podporují. Tedy pokaždé, co se aplikace spustí, je tomuto softwaru předána zpráva, kdy a na jakém PC byla aplikace spuštěna a jestli proběhla v pořádku či nikoliv. [9]

3.5 OpenCL

3.5.1 Informace

OpenCL je standard pro multiplatformní a paralelní programování heterogenních počítačových systémů. Umí využívat jak GPU, APU, tak dokonce i CPU a mnoho dalších. Jazyk OpenCL výrazně zvyšuje rychlost v širokém spektru aplikací, např. na náročné výpočty, profesionální kreativní nástroje, vědecký software nebo školení a odvozování neuronových sítí. [10]

OpenCL zrychluje aplikace a programy tak, že rozloží náročné výpočty až na několik tisíc částí, tzv. vlákna. Těchto vláken má většina GPU několik stovek, ty nejvýkonnější desítky tisíc. Jedno tohle vlákno je sice pomalejší než vlákno na CPU, ale díky velkému množství se výpočtů provede najednou mnohem více. V jednom PC se může nacházet i více GPU, zde záleží na počtu slotů PCIe na základní desce, kde se grafické karty zapojí. Většina základních desek podporuje 1 - 4 slotů, ale existují i desky, které těchto slotů mají 8. Počet výpočtů za stejnou dobu se zde násobí, tedy při osmi GPU můžeme dosáhnout osm krát více výpočtů za stejnou dobu. [10]

3.5.2 Jazyk

OpenCL využívá standardu C99, který byl vyvinut ze známého programovacího jazyku C. Jazyk má v sobě zahrnutých spoustu relačních, matematických, geometrických, synchronizačních funkcí

nebo např. čtení a zápis obrázku. Obsahuje skalární i vektorové datové typy, pointery a obrázky. Podporuje i číselné datové typy s plovoucí desetinnou čárkou jako jsou float nebo double. [11]

3.5.3 Důležité pojmy a identifikátory

Jeden z důležitých indentifikátorů je _kernel, který deklaruje danou funkci jako kernel, což znamená, že je viditelná pro host kód (většinou se jedná o CPU). [11]

[11] **Kvalifikátory adresního prostoru:**

- _global - globální, sdílená paměť všemi vlákny,
- _local - lokální pamět, sdílená pouze vlákny v lokální skupině,
- _constant - paměť pouze pro čtení daných konstant,
- _private - jej uvidí každé vlákno zvlášť.

Výchozí nastavení pro všechny proměnné, pokud nejsou definované, je _private.

[11] **Funkce v pracovní skupině:**

- get_global_id() - získání unikátního ID vlákna,
- get_local_id() - získání unikátního ID vlákna v rámci lokální skupiny.

[11] **Synchronizační funkce:**

- barrier(cl_mem_fence_flags flags) - všechna vlákna v pracovní skupině musí spustit bariérní funkci předtím, než jakékoliv vlákno může pokračovat,
- mem_fence(cl_mem_fence_flags flags) - čeká, dokud všechny operace čtení a zápisů do lokální nebo globální paměti jsou viditelná v pracovní skupině.

3.5.4 Omezení

Většina technologií má svá omezení. Není zde povolená rekurze nebo pointery na funkce. Pointery na pointery jsou povoleny pouze v kernelu, ale ne jako argument kernelu. Bitová pole nejsou podporována. Deklarace délky pole pomocí proměnné není podporována, musí být dána konstantou. Struktury a ostatní datové typy musí být definovány jak v host tak device kódu. [11]

3.5.5 Postup OpenCL

V následující krátké ukázce kódu lze vidět vytvoření nového OpenCL programu a jeho kernelu v jazyce C# s použitím OpenCL rozšíření Cloo.

```
cpl = new ComputeContextPropertyList(ComputePlatform.Platforms[0]);
context = new ComputeContext(ComputeDeviceTypes.Gpu, cpl, null, IntPtr.Zero);
Dev = context.Devices[0];
program = new ComputeProgram(context, new string[] { Kernel });
program.Build(null, null, null, IntPtr.Zero);
commands = new ComputeCommandQueue(context, dev,
    ComputeCommandQueueFlags.None);
events = new ComputeEventList();
kernel = program.CreateKernel("MultiplyNumbers");
```

Výpis 3.1: Ukázka vytvoření OpenCL programu a jeho kernelu v jazyce C#

Vytvoříme si novou ComputePlatformu a ComputeContext, kde vybereme zařízení Dev. Je zde na výběr GPU, CPU nebo All, což znamená všechna zařízení. Poté si vytvoříme nový program, do kterého vložíme řetězec OpenCL programu, který je napsaný v datovém typu string. Poté se program zkompiluje, vytvoříme instanci třídy ComputeCommandQueue a vytvoříme kernel funkci, kde vložíme název kernel funkce v řetězci.

V následující ukázce kódu lze vidět vytvoření dat, jejich odeslání, spuštění kernelu a zpětné načtení výsledků.

```
int[] numbers = Enumerable.Range(2, 1000000).ToArray(), results = new int
    [numbers.Length];
ComputeBuffer<int> Numbers = new ComputeBuffer<int>(
context, ComputeMemoryFlags.ReadWrite | ComputeMemoryFlags.UseHostPointer,
    numbers);
kernel.SetMemoryArgument(0, Numbers);
commands.Execute(kernel, null, new long[] { numbers.Length }, null, events);
commands.ReadFromBuffer(Numbers, ref results, false, events);
Numbers.Dispose();
```

Výpis 3.2: Ukázka vytvoření dat, spuštění programu a čtení výsledku

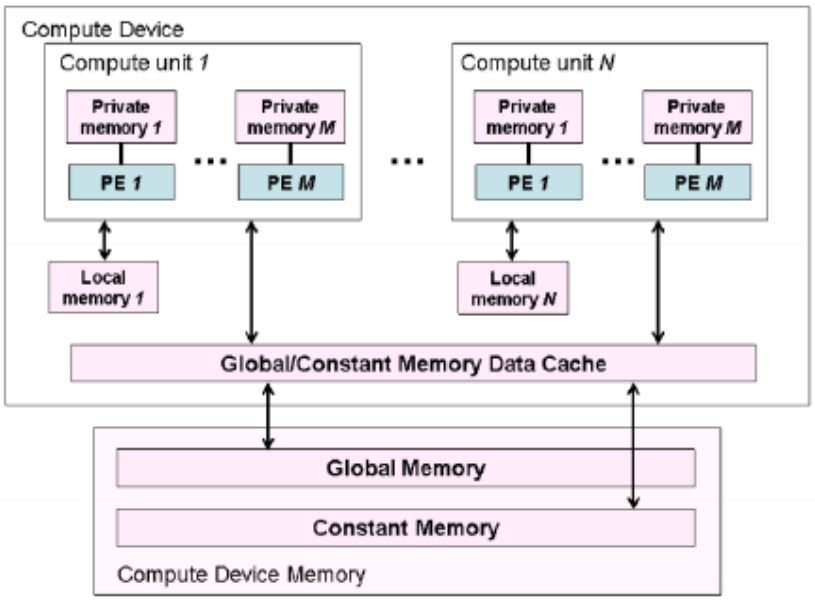
V prvním řádku se vytvoří pole čísel od 2 do 1 mil. a pole výsledků. V druhém až čtvrtém řádku se vytvoří ComputeBuffer, který slouží k uložení tohoto pole do paměti GPU. Jsou zde možnosti pouze číst, pouze zapisovat nebo čtení i zápis zároveň. V následujícím řádku se pole přidá jako argument kernelu. Další řádek spustí kernel. Zde je důležitý parametr numbers.Length, který udává délku pole. To znamená, že se kernel spustí přesně tolikrát, jak je dlouhé pole dat. Předposlední řádek se stará o načtení dat zpět z paměti GPU do paměti RAM, kde s nimi už může pracovat CPU. Data se uloží do proměnné results. Poslední řádek se stará o uvolnění paměti na GPU.

Výhody OpenCL [12]

- Velice efektivní a paralelní programovací technologie.
- Je nezávislý na OS. Lze s ním pracovat na OS Windows, Linux.
- Není omezen na výrobce komponent. Umí pracovat s produkty od AMD, NVIDIA a Intel.

Nevýhody [12]

- V dnešní době se již tolik nepoužívá jako např. technologie CUDA.
- Je založený na jazyku C99, nejsou zde třídy a templaty.
- Kernel musí být napsán v řetězci, což komplikuje čitelnost kódu.



Obrázek 3.1: Paměťový model OpenCL [13]

Na Obrázku 3.1 lze vidět paměťový modul OpenCL. Výpočetní zařízení (Compute Device) se skládá z výpočeních jednotek (Compute units) neboli pracovních skupin (Work groups), kde každá z nich má svou privátní paměť a lokální paměť. Pracovní skupiny se dále dělí na jádra, kde každé z jader počítá svůj PE (Processing Element), tzn. zpracovávaný prvek. Tyto skupiny poté sdílí globální paměť, kde lze využívat globální proměnnou a konstantní paměť, kterou lze číst z konstant. [13]

Kapitola 4

Zadané úlohy a jejich řešení

Tato kapitola obsahuje seznam a podrobnější popis úloh, které mi byly přiděleny na odborné praxi. V našem závodě se nachází mnoho serverů, kde se vytváří každý den logy. Tyhle logy se musí po dobu jednoho roku zachovat, pokud by se někdo potřeboval vrátit na určitý den a podívat se, co se daný den stalo. Jsou to např. transakce do databáze, alarmy, zpracování událostí, denní joby atd. Logy se musí zapisovat, kopírovat a mazat ručně. Mého konzultanta tedy napadlo tento proces zautomatizovat a dělat všechny tyhle operace pomocí spuštění jednoho programu.

4.1 Program PCChecker

V první úloze jsem měl za úkol napsat konzolovou aplikaci v jazyce C# pro práci se soubory. Poté jsem měl za úkol tuto aplikaci rozšířit o funkci kontroly disků a určit jejich momentální volnou kapacitu.

4.1.1 Práce se soubory

Jednou z funkcí programu PCChecker je práce se soubory. Umí základní operace jako jsou: mazání, kopírování, přesunutí, zazipování a přesunutí zároveň. V následující krátké ukázce kódu si ukážeme konfigurační soubor programu, jeho funkce a základní parametry.

```
<add key="File1_From" value="*" />
<add key="File1_To" value="2020-01-25 11:00" />
<add key="File1_FromTo" value="false" />
<add key="File1_WorkDay" value="today" />
<add key="File1_MinusDays" value="30" />
<add key="File1_SrcPath" value="C:\Users\Test\Desktop\TestDir" />
<add key="File1_DstPath" value="C:\Users\Test\Desktop\Destdir" />
<add key="File1_FileName" value="TestovaciZip" />
```

```
<add key="File1_FileNameWithDate" value="true" />
<add key="File1_ZipRewrite" value="false" />
<add key="File1_Operation" value="zip" />
<add key="File1_FileMask" value="*acro/YY/MMm*.*" />
```

Výpis 4.1: Ukázka konfiguračního souboru PCChecker 1.

První a druhý parametr je rozmezí stáří souboru od do. Tuhle funkci je nejprve potřeba zapnout nastavením třetího parametru na hodnotu "true". Při použití této funkce se datum do těchto proměnných píše ve formátu "yyyy-MM-dd HH:mm", kde jednotlivá písmena postupně znamenají: rok, měsíc, den, hodina, minuta. U parametru from je možnost zadat "*", která znamená rok 0001.

Na čtvrtém řádku je parametr WorkDay, který značí datum, se kterým pracujeme a které bude obsaženo v názvu souboru. Pro tuto variantu musí být zapnutá funkce FileNameWithDate. Lze jej nastavit na "today", která je datum dnešního dne, "yesterday" znamená včerejší den, "FirstDayThisYear" pro první den tohoto roku, "LastDayLastYear" pro poslední den minulého roku nebo zadat jakýkoliv den ve formátu yyyy-MM-dd.

Parametr MinusDays je počet dní, o kolik se má WorkDay zmenšit. Parametr SrcPath značí zdrojovou cestu, tedy v jakém adresáři se operace provedou. DstPath je adresář, kam se konečné soubory přesunou.

Klíč Filename slouží pro určení názvu zip složky a FileNameWithDate pro přidání data WorkDay k tomuto názvu. Název by tedy nebyl "TestovaciZip", ale "TestovaciZip20211803".

Funkce ZipRewrite slouží pro zapnutí přepisování. Pokud je nastaven na "true", přidá a přepíše soubory, jinak soubory smaže a pracuje pouze s aktuálními.

Parametr Operation slouží pro určení operace. Tyhle operace jsou zmíněny v kapitole 4.1.1.

Poslední proměnná FileMask je maska, podle které vyhledáváme všechny soubory, které tento název obsahují. "*" znamená cokoliv, "." odděluje název od přípony. Datum se bere z proměnné WorkDays. Např. pokud WorkDays bude mít v kódu zmíněnou hodnotu, maska bude "*acro2113m*.*".

V následující ukázce kódu lze vidět příklad jedné z operací.

```
case "ZIP":
   using (ZipArchive zip = ZipFile.Open(param.DestPath + param.FileName +
       ".zip", ZipArchiveMode.Update))
      {
         if (param.ZipRewrite.ToUpper() == "FALSE")
         {
         DeleteAllFromZip(zip);
         }

         foreach (string i in files)
         {
```

```
            var info = new FileInfo(i);
            if (param.ZipRewrite.ToUpper() == "TRUE")
            {
            DeleteFromZip(zip, info);
        }

        if (param.FromTo.ToUpper() == "TRUE" && info.LastWriteTime >=
            param.HowOldFrom && info.LastWriteTime <= param.HowOldTo)
            zip.CreateEntryFromFile(i, info.Name);
        if (param.FromTo.ToUpper() != "TRUE")
            zip.CreateEntryFromFile(i, info.Name);
    }
```

Výpis 4.2: Ukázka operace PCChecker

Větev switche Zip se spustí, jestliže konfigurační parametr FileX_Operation v konfiguračním souboru aplikace je nastaven na "zip". Ve druhém řádku se otevře daný zip archiv. Využívá se zde knihovny System.IO.Compression, která je využívána pro kompresi souborů do .zip archivu a práci s nimi. Pokud je v první podmínce konfigurační parametr nastaven na "false", všechny soubory se z archivu vymažou a pracuje se pouze s novými. V opačném případě se přepisování povolí.

Cyklus foreach projde všemi soubory, které se nachází na specifikované cestě s danou maskou a pokud je povoleno přepisování, soubory se stejným názvem se přepíšou.

V posledních dvou podmínkách lze vidět kontrolu, zda využíváme funkce FromTo. Pokud ano, zazipují se jen soubory, které splňují podmínku, že jejich datum poslední úpravy souboru je mezi hodnotami param.HowOldFrom a param.HowOldTo. Proměnná info.LastWriteTime toto datum určuje.

4.1.2 Práce s disky

Druhou důležitou funkcí je práce s disky. Je zde mnoho funkcí, které lze využívat. Aplikace se spouští jednou denně téměř na každém serveru závodu vysokých pecí a má za úkol zkontrolovat aktuální volné místo na každém z disků. Pokud je volné místo menší než je nastavené minimum, program automaticky zasílá email všem emailovým adresám, které jsou zde zadány.

```
<add key="disk1" value="C:" />
<add key="disk2" value="D:" />
<add key="disk3" value="E:" />
<add key="C:" value="400000000" />
<add key="D:" value="400000" />
<add key="E:" value="100000" />
```

```
<add key="EmailAddress" value="test@test.cz;test2@test.cz" />
<add key="EmailSubject" value="Warning disk space on PC " />
<add key="DefaultMinSpace" value="100" />
<add key="AutoMessageCZ" value="Tento email byl vygenerovan automaticky,
   neodpovadejte na nej." />
<add key="AutoMessageEN" value="This email was generated automatically, please
   do not reply." />
<add key="UseRabbit" value="True" />
<add key="DebugInfo" value="false" />
```

Výpis 4.3: Ukázka konfiguračního souboru PCChecker 2.

V prvním až třetím řádku kódu se nachází disky a jejich písmena. Těchto disků se může zadat libovolný počet. Ve čtvrtém až šestém řádku se nachází disky a jejich minimální hodnoty v MB. Pokud je volné místo disku menší než tato hodnota, odesílá se email, ve kterém se píše, že je potřeba zkontrolovat daný disk na daném PC.

Parametr EmailAddress obsahuje jednu nebo více emailových adres, na které se má v případě malého volného místa na disku odeslat email. Pokud se jedná o více než jednu adresu, jednotlivé adresy se oddělují středníkem.

Proměnná EmailSubject je předmět emailu. DefaultMinSpace je výchozí hodnota zapsaná v MB, která se nastaví na disk, pokud nemá definovanou minimální hodnotu.

AutoMessageCZ a AutoMessageEN slouží pro konečnou větu v těle emailu. Tato zpráva je napsaná v českém a anglickém jazyce a informuje příjemce, že na email nemá odpovídat.

Funkce UseRabbit slouží pro použití technologie RabbitMQ [9], která je zde využívána k odesílání zpráv, zda aplikace proběhla úspěšně, případně k uchování chybových hlášek.

DebugInfo je čistě informativní funkce, která vypíše do konzole aktuálně zapnuté instance programu PCChecker všechny zadané parametry v konfiguračním souboru a jejich hodnoty.

```
using (var connection = new SqlConnection(connectionString))
   {
   using (var command = new SqlCommand("[msdb].[dbo].[sp_send_dbmail]",
       connection))
      {
         connection.Open();
         command.CommandType = CommandType.StoredProcedure;
         command.Parameters.AddWithValue("@recipients", recipients);
         command.Parameters.AddWithValue("@subject", string.Format("{0}",
            ticketCode));
         command.Parameters.AddWithValue("@body", body);
         command.Parameters.AddWithValue("@body_format", "HTML");
```

```
            command.Parameters.AddWithValue("@profile_name", "MES12_PCChecker");
            command.ExecuteNonQuery();
        }
    }
```

Výpis 4.4: Ukázka funkce pro odeslání emailu

Zde je využita databáze, do které se připojíme pomocí řetězce connectionString. Konkrétně jde o volání uložené procedury sp_send_dbmail. Proceduře předáme pět různých parametrů.

Prvním je @recipients, což jsou emailové adresy zadané v parametru EmailAddress v konfiguračním souboru. Druhým parametrem je @subject, který je předmět emailu a je taktéž zapsán v konfiguračním souboru. Dalším parametrem je @body, což je tělo emailu. Poté následuje @body_format, který udává, v jakém formátu je zpráva vytvářena. Posledním parametrem je @profile_name, který udává odesílatele emailu. V posledním kroku se spustí procedura.

4.2 Seznámení s hlavním projektem a jeho analýza

Po dokončení aplikace pro správu souborů a kontrolu disků bylo mým úkolem se seznámit s hlavním projektem Technologická a finanční kalkulace vsázky pro výrobu aglomerátu a surového železa. Jedná se o poměrně složitý projekt, na kterém pracoval celý tým přibližně půl roku.

4.2.1 Analýza

V procesu analýzy hlavního projektu jsme řešili, jaké technologie se na projekt budou využívat a v jaké míře. Také jsme řešili, jak by měla kalkulace vypadat, jaké paramatry by měla obsahovat atd. První náhled výpočtu a jeho vzorce byl vytvořen v programu Microsoft Excel, podle kterého se poté tvořil výsledný program. Tomuto problému se věnoval zbytek týmu. Mým úkolem bylo zjistit a popřípadě otestovat využití GPU k výpočtům společně s programovacím jazykem C#.

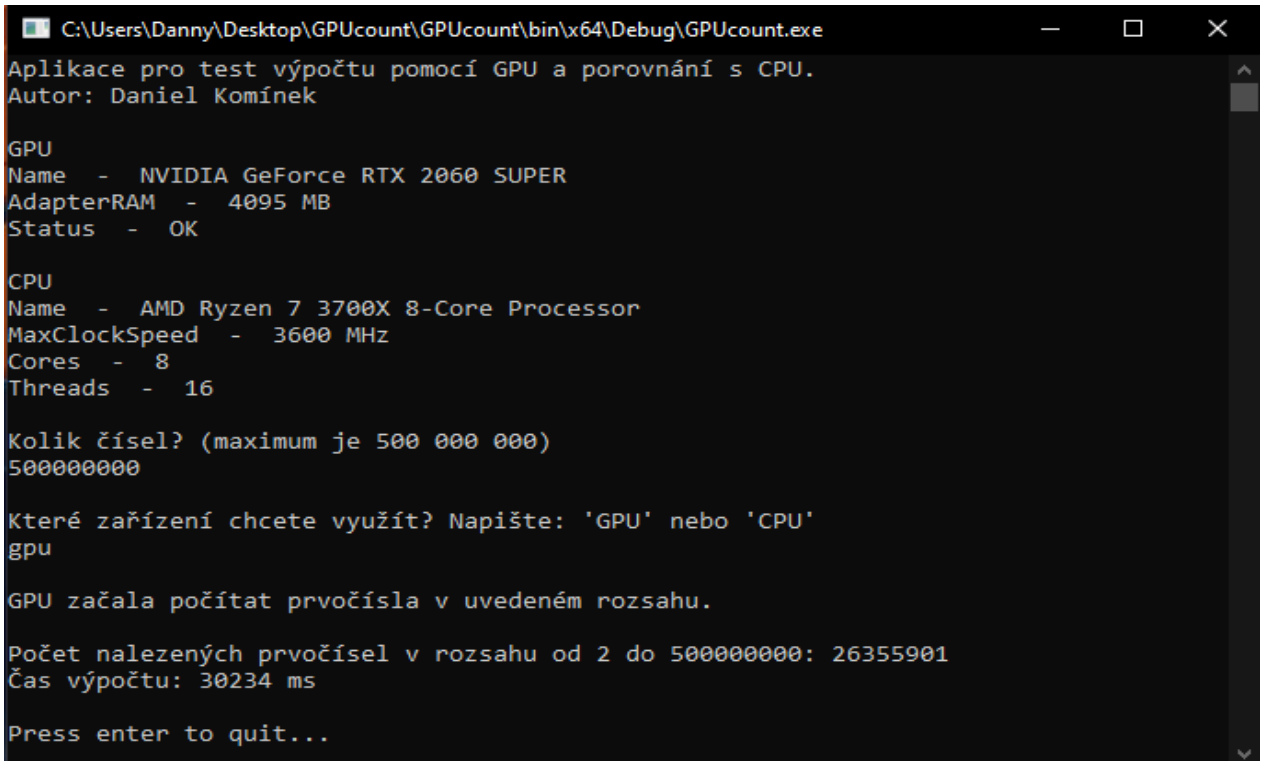
4.3 Testovací program GPUCount

Existuje rozšíření Cloo.clSharp, pomocí kterého lze využít technologii OpenCL právě ve spojení s programovacím jazykem C#. Právě toto rozšíření je v tomto i hlavním projektu využito. Autory tohoto rozšíření jsou Fatjon Sakiqi a Alexander Hildebrand a lze si jej nainstalovat do vývojového prostředí Microsoft Visual Studio.

Jak již bylo zmíněno v odstavci 4.2.1, tak mým dalším úkolem bylo vytvořit konzolovou aplikaci, která zjistí veškeré důležité informace o PC, na kterém aktuálně běží a podle toho jej využije. Výpočet je proveden jak na procesoru, tak i na grafické kartě. Za testovací algoritmus pro porovnání výpočtů jsem zvolil vyhledání prvočísel v rozsahu od 2 do čísla N, které zadá uživatel. Algoritmus

sice na první pohled vypadá jednoduše, avšak čím větší je číslo N, tím je více výpočtů v jedné iteraci provedeno.

4.3.1 Ukázka programu GPUCount



Obrázek 4.1: Ukázka programu GpuCount

Program vypíše základní informace o GPU - její úplný název, její velikost RAM a její status, zda je v pořádku a běží. Také vypíše parametry CPU, kde je to rovněž název, jeho frekvenci a počet jader a vláken.

Pokud by PC obsahoval více než jednu GPU, vypsaly by se postupně pod sebe: GPU1, GPU2 atd. Následně si uživatel může zvolit horní limit, do kterého se mají prvočísla nalézt. Pokud zadáme 0,5 mld., jak lze vidět na Obrázku 4.1, budou to čísla od 2 do 0,5 mld. postupně uložená do pole. Do pole jsou ukládána z důvodu, že GPU s jiným datovým typem např. seznamem pracovat neumí. Limit 0,5 mld. je nastaven z důvodu, že tato konkrétní grafická karta má 4GB RAM a právě 0,5 mld. čísel uložených v poli integerů zabírá 4GB paměti. Pokud bychom tedy chtěli spočítat více čísel, museli bychom toto pole rozdělit na dvě či více částí. Dále si uživatel může vybrat, zda chce prvočísla spočítat pomocí CPU nebo GPU. V programovacím jazyce C# je velikost jednoho objektu omezená na 2 GB. Pokud chceme alokovat objekt větší než 2 GB, je potřeba povolit alokaci větších objektů pomocí následujícího kódu v konfiguračním souboru aplikace.

```
<runtime>
   <gcAllowVeryLargeObjects enabled="true" />
</runtime>
```

Výpis 4.5: Povolení alokace objektů větších než 2 GB

V předposledním řádku konzole je vypsán uvedený rozsah a poté počet prvočísel, kolik jich bylo v uvedeném rozsahu nalezeno. Poslední řádek vypíše celkový strávený čas v milisekundách.

V tomto případě výpočet na GPU trval 30,2 sekund. Procesoru paralelně na všech šestnácti vláknech stejný počet čísel zabral 307 sekund. V tomto případě je to tedy více než 10x rychlejší. Výpočty na grafických kartách se vyplatí při větším počtu čísel. Čím větší počet operací, tím více se vyplatí. Pokud bychom chtěli např. vypočítat 1 mil. čísel, tak GPU to zabere 0,2 sekund, zatímco CPU pouze 0,1 sekund viz Tabulka 4.2. Hlavním důvodem je přenos dat mezi pamětí CPU a GPU.

4.3.2 Výpočet na CPU

```
public static bool IsPrime(int n)
   {
      int upperl = (int)Math.Sqrt((float)n);
      for (int i = 2; i <= upperl; i++)
      {
         if (n % i == 0)
            return false;
      }
      return true;
   }
   List<int> primes = new List<int>();
   Parallel.ForEach(numbers, (item) =>
   {
      if (IsPrime(item))
      {
         lock (lockMe)
         {
            primes.Add(item);
         }
      }
   });
```

Výpis 4.6: Výpočet prvočísel na CPU

V první části kódu můžeme vidět funkci na kontrolu, zda je dané číslo prvočíslem či nikoliv. Pokud je prvočíslem, funkce vrátí true, v opačném případě vrátí false. Parametrem funkce je číslo N, které chceme ověřit.

V druhé části kódu lze vidět paralelní procházení pole, které jsme vytvořili na začátku programu, jak lze vidět na Obrázku 4.1. Jedná se o třídu Parallel, která na zpracování příkazu využije všech vláken, které má CPU k dispozici. Pokud by zde nebyla využita tato třída a použili bychom klasický foreach, vše by se vykonalo sekvenčně na jednom vláknu. Na tomto konkrétním CPU by tento čas byl více než osm krát větší. Platí zde stejné pravidlo jako při využití grafických karet, tedy využití paralelního zpracování se vyplatí při vyšším počtu operací. Ve funkci IsPrime ve Výpisu 4.6 se paralelní verze cyklu for nevyplatí, jelikož zde není takové množství operací a bylo by to spíš na škodu.

Foreach projde každé číslo v poli a pomocí výše zmíněné funkce zjistí, zda je prvočíslem či nikoliv. Pokud prvočíslem je, přidá je do seznamu. Je zde zámek, který zaručí, aby ve stejném okamžiku k seznamu přistupovalo pouze jedno vlákno a nedošlo tak ke zbytečným potížím v ukládání dat do proměnné.

4.3.3 Výpočet na GPU

```
kernel void GetIfPrime(global int* message)
   {
      int index = get_global_id(0);
      int upperl = (int)sqrt((float)message[index]);
      for (int i = 2; i <= upperl; i++)
         {
            if (message[index] % i == 0)
            {
               message[index] = 0;
               return;
            }
         }
   };
```

Výpis 4.7: Výpočet prvočísel na GPU [14]

Zde vidíme stejný algoritmus jako ve Výpisu 4.6, ale ve verzi pro grafickou kartu. Funkce je tedy napsaná v jazyce C99, který je hodně podobný jako jazyk C. Jde o tzv. kernel.

Parametrem kernelu je pointer na čísla, ze kterého se čte i zapisuje. Jedná se tedy o stejné vytvořené pole čísel jako v Obrázku 4.1. Do proměnné index se pomocí funkce get_global_id() uloží unikátní číslo pro každé vlákno, tedy např. vlákno 0 bude mít index 0, vlákno 32 bude mít index 32

atd. Díky tomuto můžeme v poli k jednotlivým hodnotám přistupovat paralelně. Poté algoritmus pokračuje stejně, jako v případě CPU, jen se zde prvočísla ukládají rovnou do stejného pole. Pokud prvočíslem není, uloží se zde 0, pokud je, číslo zůstane. Konečné pole tedy bude obsahovat čísla: 2, 3, 0, 5, 0, 7, 0, 0, 0, 11 atd.

4.3.4 Porovnání výpočtů

Tabulka 4.1: Jednotlivé sestavy

Testovací sestavy			
	CPU	**GPU**	**RAM**
PC1	AMD Ryzen 7 3700x 4,4 GHz, 8 jader, 16 vláken	Nvidia Geforce RTX 2060 Super 2176 CUDA jader, 8GB GDDR6	16GB DDR4 3,2 GHz
PC2	Intel Core i3-10100 4,2 GHz, 4 jádra, 8 vláken	Intel UHD Graphics 630 24 jader, využívá CPU RAM	8GB DDR4 2,4 GHz
PC3	Intel Core i7-7820HQ 3,9 GHz, 4 jádra, 8 vláken	Nvidia Quadro M2200 1024 CUDA jader, 4GB GDDR5	8GB DDR4 2,4 GHz
PC4	Intel Core i5-7300U 3,5 GHz, 2 jádra, 4 vlákna	Intel HD Graphics 620 24 jader, využívá CPU RAM	8GB DDR4 2,4 GHz
PC5	AMD Ryzen 5 3500U 3,7 GHz, 4 jádra, 8 vláken	AMD Radeon Vega 8 Graphics 512 jader, využívá CPU RAM	8GB DDR4 2,4 GHz
PC6	Intel Core i5-8250U. 3,4 GHz, 4 jádra, 8 vláken	Nvidia Geforce MX130 384 CUDA jader, 4GB GDDR5	8GB DDR4 2,4 GHz

Tabulka 4.2: Porovnání časů výpočtů

Porovnání výpočtů GPU vs CPU v sekundách								
	1 mil.		**100 mil.**		**250 mil.**		**500 mil.**	
	CPU	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**
PC1	0,1	0,2	30	3	112	11	307	30
PC2	0,1	0,3	34	33,2	121	120	-	-
PC3	0,2	0,2	62,3	23,1	221,1	83,5	-	-
PC4	0,3	0,3	83,7	25	322,2	90,5	-	-
PC5	0,2	0,7	78,4	53,2	287	-	-	-
PC6	0,2	1,3	43,1	98,8	168,9	-	470,5	-

V Tabulce 4.1 je seznam všech využitých PC sestav a notebooků a jejich důležité parametry pro tento výpočet. U CPU je to název, frekvence v GHz, počet jader a počet vláken. Důležité parametry GPU jsou její název, počet jader nebo výpočetních jednotek, a pokud je to dedikovaná grafická karta, tak je zde uveden typ a velikost její paměti. V případě integrovaných GPU je využívána paměť RAM. U operační paměti RAM je důležitá její velikost, typ a frekvence, na které běží. První dvě zařízení jsou stolní počítače, ostatní jsou notebooky.

V Tabulce 4.2 jsou všechny PC a jednotlivé časy výpočtu uspořádány podle počtu testovaných čísel. Jsou zde sestavy a notebooky ve všech výkonnostních a cenových relacích. Zde jde velice dobře vidět, kdy se vyplatí využít GPU na výpočty a kdy ne, a to i v případě, že je GPU integrovaná. I ta je totiž ve větším množstvím operací většinou rychlejší než CPU. Výjimkou je PC2, který je kombinací rychlého CPU a velmi slabé integrované GPU, proto jsou čísla skoro totožná. Pokud se podíváme na 1 mil. čísel, tak je procesor většinou rychlejší, jelikož se jedná o malé množství operací.

Symbol pomlčky symbolizuje, že zde nebyl výpočet možný z důvodu hardwarových omezení. Pokud totiž alokujeme data na integrovanou GPU, tato data jsou alokovány celkem dvakrát. Poprvé na CPU a podruhé jsou zkopírovana i na GPU. Tedy v případě 500 mil. čísel je velikost dat přibližně 4GB. U integrované GPU by na to bylo zapotřebí až 8GB paměti RAM, zatímco dedikovaná GPU má svou vlastní paměť.

4.3.5 Využití více GPU současně

Dalším úkolem bylo zjistit, jak použít OpenCL s více grafickými kartami současně a to byla také další možnost, jak ještě výpočet prvočísel zrychlit. Bohužel nemám k dispozici PC, který by disponoval dvěma či více GPU. Firma mi poskytla na dálkové připojení PC na testování, který má dvě GPU. Jednu integrovanou a druhou dedikovanou. Bohužel tyto GPU nebyly výkonnostně srovnatelné, ale aspoň jsem měl možnost si vyzkoušet, jak taková práce s více GPU funguje a ověřit si, zda-li je to vůbec možné. I když je jedna GPU výkonnější než druhá, je možné dosáhnout zrychlení při využití obou současně.

Aby paralelní výpočet na více GPU současně fungoval správně a nejefektivněji, je potřeba celkovou práci rozdělit na tolik částí, kolik máme GPU. V tomto případě se jedná o dvě GPU s různým výkonem. Tedy výpočet rozdělíme na dvě části. Bude se jednat o jednu menší část, kterou přidělíme slabší GPU a o druhou větší část, kterou přidělíme té silnější. Tato forma rozdělení na dvě části odlišné velikosti je zde použita, aby došlo k přiblížení obou časů výpočtů grafických karet v co největší možné míře.

Testoval jsem tři různé možnosti, po kterých jsem našel ideální poměr rozdělení. Nejprve jsem vyzkoušel rozdělení na dvě stejně velké části, protože jsem doposud neznal výkon obou karet. Výpočet zde trval sice kratší dobu, než když by jej počítala sama slabší GPU, ale trval déle, než by to trvalo výkonnější GPU. Poté jsem zkusil rozdělení v poměru 1:3, kde slabší karta počítala jednu čtvrtinu všech čísel a výkonnější zbylé tři čtvrtiny. Zde byl čas viditelně kratší, ale přišel jsem na to, že výkonnější GPU počítá svou část déle. Vyzkoušel jsem tedy ještě rozdělení 1:2, které je lepším řešením. Rozdíl obou časů je zanedbatelný. V Tabulce 4.3 lze vidět dobu výpočtů, když jej GPU počítá sama a pod nimi využití obou současně.

Tabulka 4.3: Využití dvou GPU současně

(a) Testovací PC

CPU	Intel Core i5-7500, 3,8 GHz 4 jádra, 4 vlákna
GPU1	AMD Radeon R7 450 512 jader, 4GB GDDR5
GPU2	Intel HD Graphics 630 24 jader, využívá RAM
RAM	16 GB, 2,4 GHz

(b) Porovnání

Jedna GPU vs obě současně		
GPU, poměr	**100 mil.**	**250 mil.**
1.	89,1	346
2.	27	97,6
1. + 2., 1:1	32	125
1. + 2., 1:3	23	83
1. + 2., 1:2	21	76

Podle těchto čísel lze předpokládat, že pokud bychom měli např. dvě stejně výkonné grafické karty, dostali bychom se na dvojnásobek výpočtů za stejný čas. Pokud by grafické karty byly čtyři, rozdíl by byl čtyřnásobný atd.

4.4 Výpočet vsázky

Tým byl velmi překvapen, jak velkého zrychlení lze dosáhnout při zapojení GPU do výpočtu. Můj hlavní úkol spočíval v přepisu složitějšího algoritmu pro výpočet vsázky na GPU, který byl dosud paralelně počítán pomocí CPU a který byl již zmíněný v kapitole 4.2. Zdrojový kód k projektu je přiložen v příloze A této práce.

4.4.1 Informace o projektu

Projekt je rozdělen na dvě části. Jedná se o klienta a server. Projekt BF_ChargeCalculationWCF je server, který má definované public metody, které poskytuje navenek svému klientovi. Je zde důležitá metoda GetValidChargeCalculationResults, kterou volá klient. Tato metoda se stará o zpracování vstupních parametrů, které uživatel do výpočtu zadá a vytváří zde struktury a třídy, které jsou k výpočtu potřebné. Stará se také o logování do souboru, měření času výpočtu a volání metody AllChargeCalculationParallelForeach, která se vstupními parametry provede výpočet pomocí CPU. Mým dalším úkolem bylo přepsat tuto metodu na GPU.

4.4.2 Vstupní a výstupní parametry

Uživatel může zadat mnoho parametrů. Nejdůležitější parametr, který udává, kolik výpočtů bude provedeno, je počet materiálů. Celou dobu jsem pracoval se sedmi nebo osmi materiály. Lze však zadat libovolný počet materiálů. Každému z těchto materiálů se ještě volí parametry: MinimumQuantity, MaximumQuantity, StepForQuantity a UnitPrice. Minimální a maximální kvantita určuje rozsah a parametr StepForQuantity určuje krok. Např. pokud bychom měli materiál, který

by měl minimální kvantitu 40, maximální 60 a krok by byl 2, tak by bylo u tohoto materiálu celkově 11 kombinací. Kvantita by byla 40, 42, 44 atd. až do 60. Takto je to vyhodnoceno pro každý vytvořený materiál s těmito vybranými parametry. Všechny kombinace materiálů jsou uloženy do 2D seznamu a funkce GetAllCombinations se poté postará o kombinaci každého materiálu s každým tzv. kartézský součin, aby bylo možné výpočet provést pro každou kombinaci. Jsou zde dvě třídy, které obsahují uživatelem zadané parametry.

Třída UserChargeSettings obsahuje vstupní parametry RequestSi, RequestC, RequestBasicity a RequestCokeAsh.

Třída ChargeRequestParameters obsahuje výstupní parametry, které udávají jaké má být minimum a maximum daného parametru ve výsledku. Jsou to požadavky na kvalitu vsázky. Např. chceme, aby výsledek vsázky obsahoval jen materiály, u kterých výjde basicita 1,1 až 1,2. Basicita je jeden z parametrů třídy Result. Uživatel může zadat minimum i maximum, nebo jenom jeden z nich a nebo žádný. Jsou to parametry: C, Si, Yield, ChargeRichness, Basicity, Mn, MgO, SlagQty, CokeSpecificConsumption.

4.4.3 Generování všech kombinací zadaných materiálů

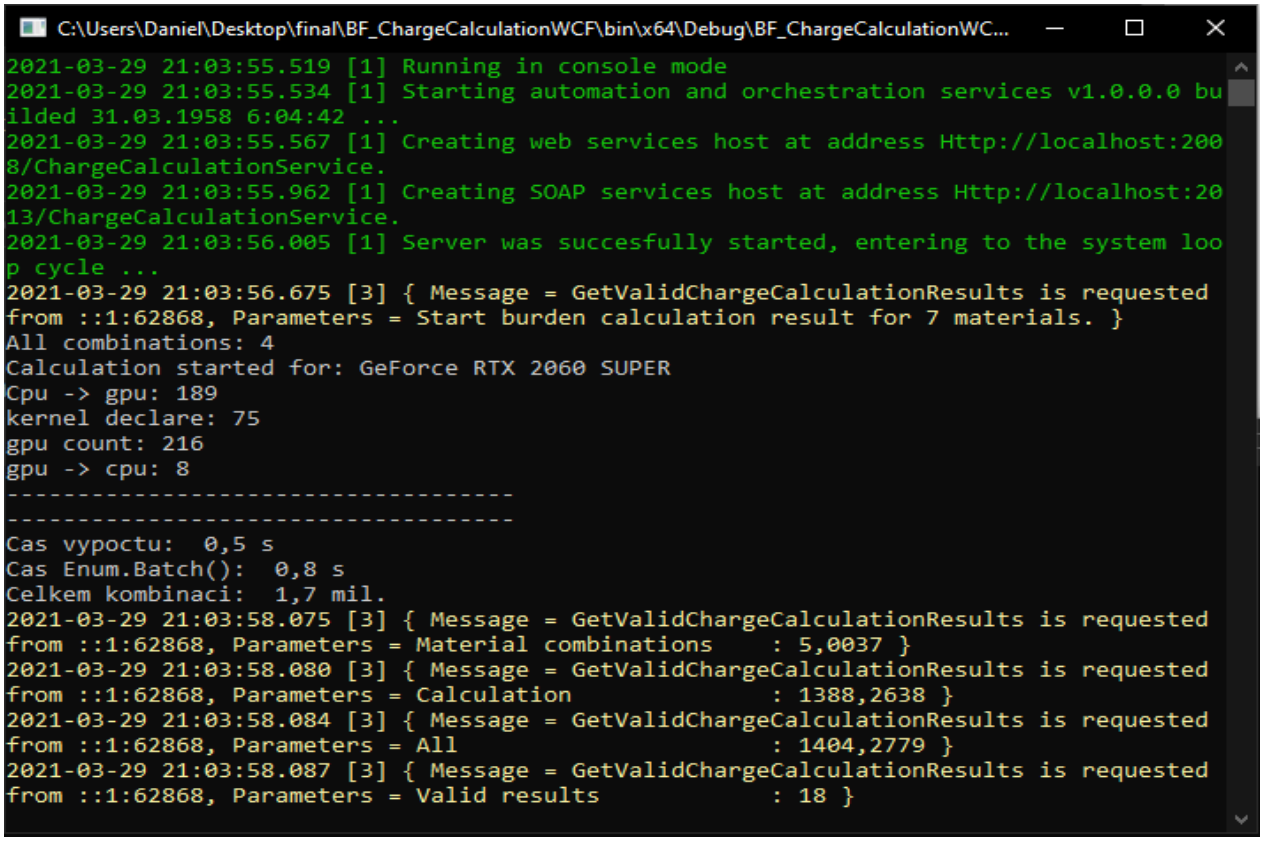
V následujícím krátkém kódu si ukážeme generaci všech kombinací pomocí technologie LINQ, která je zmíněná v kapitole 3.2. LINQ využívám, protože kombinací k výpočtu může být nekonečně mnoho a ty by tak zabíraly obrovské množství paměti. Díky LINQ se k hodnotám přistupuje pouze tehdy, když je to právě nutné a nezabírá tak zbytečně paměť RAM. Zkoušel jsem tuto variantu, kde se kombinace ukládaly do pole. 1,7 mil. kombinací zabralo přibližně 2GB RAM. Toto řešení je nepřípustné, protože běžně se počítají miliardy kombinací. Výhodou je také rychlost LINQ dotazu oproti ukládání kombinací do pole. Jedinou nevýhodou je pomalejší přistupování k datům sekvence, protože data se nikam neukládají, je potřeba je v případě nutnosti vyčíslit. V takovém velkém počtu kombinací bohužel není jiná efektivní varianta.

```
public static IEnumerable<IEnumerable<T>> GetAllCombinations<T>
   (this IEnumerable<IEnumerable<T>> sequences)
   {
      IEnumerable<IEnumerable<T>> result = new[] { Enumerable.Empty<T>() };
      foreach (IEnumerable<T> sequence in sequences)
      {
         result = from accseq in result from item in sequence select
             accseq.Concat(new[] { item });
      }
       return result;
   }
```

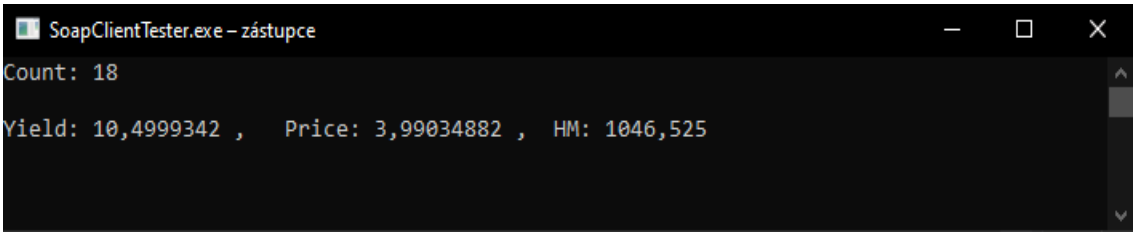
Výpis 4.8: Generace kartézského součinu materiálů

Vstupním parametrem funkce je 2D sekvence materiálů. Datový typ T je třída ChargeMaterial, která obsahuje všechny parametry materiálu. V dalším kroku se vytvoří prázdná sekvence result, do které se pomocí dotazu uloží všechny kombinace. V cyklu foreach se prochází každý z vytvořených materiálů a probíhá zde dotaz, který vytváří kartézský součin a postupně jej ukládá do sekvence result.

4.4.4 Ukázka konzolové aplikace



Obrázek 4.2: Ukázka serveru



Obrázek 4.3: Ukázka klienta

Na Obrázku 4.2 můžeme vidět serverovou část aplikace. Prvních pět zpráv vypisuje jen adresu, kde server běží, a že byl úspěšně spuštěn. Následně vidíme počet materiálů, se kterými pracujeme. Program si nejprve zjistí, jaká grafická karta je v PC dostupná a zjistí množství její paměti. Takto si program všechny kombinace inteligentně rozdělí na části, které postupně vypočítává, kde jedna část je tolik kombinací, kolik se maximálně vejde na paměť GPU. V dalších řádcích vidíme, jak probíhal samotný výpočet jedné části na GPU a jak dlouho trvaly jednotlivé funkce v ms. V tomto případě se všech 1,7 mil. kombinací vešlo do paměti GPU najednou a je tak část jenom jedna. Generace všech kombinací trvala 4ms, převod dat z CPU na GPU trval 199 ms, vytvoření OpenCL programu a jeho kernelu viz Výpis 3.1 trvalo 81 ms, výpočet na GPU trval 205 ms a převod výsledků zpět na CPU trvalo 9 ms. Pokud by částí bylo více, byly by vypsány pod sebou. Pod tím vidíme čas výpočtu, čas metody Batch a celkový počet kombinací, které počítáme. Na posledním řádku je čas spuštění celého programu.

Na Obrázku 4.3 vidíme klienta a konečný výsledek kalkulace a jeho důležité parametry. Výsledků, u kterých byly splněny všechny vstupní parametry, se našlo celkem 18.

4.4.5 Rozdělení na počet vláken

Funkce 4.4.3, která generuje všechny kombinace materiálů, využívá technologii LINQ. Ten bohužel není paralelní, a tedy vyhodnocení tohoto dotazu trvá při velkém počtu kombinací poměrně dlouho. Jeho paralelní verze PLINQ zde bohužel nefunguje. Abych tuto operaci zrychlil, rozhodl jsem se kombinace rozdělit na počet částí, kolik má CPU vláken. Každé vlákno tedy pracuje se svou částí. O tuto práci se stará metoda Batch, která rozdělí sekvenci všech kombinací na počet částí podle GPU a následně tuto část ještě rozdělí na počet vláken, které má CPU k dispozici. V následujícím fragmentu kódu si tuto metodu ukážeme.

```
public static IEnumerable<IEnumerable<TSource>> Batch<TSource>
   (this IEnumerable<TSource> source, int size)
   {
      TSource[] bucket = null;
      var count = 0;
      foreach (var item in source)
      {
         if (bucket == null)
            bucket = new TSource[size];

         bucket[count++] = item;
         if (count != size)
            continue;
```

```
            yield return bucket;

            bucket = null;
            count = 0;
        }
        if (bucket != null && count > 0)
            yield return bucket.Take(count);
    }
```

Výpis 4.9: Rozdělující metoda na části podle počtu vláken [15]

Vstupními parametry funkce je sekvence, kterou chceme rozdělit, a velikost jednotlivých částí. Vytvoří se pole bucket, do kterého se pouze po dobu nutnosti ukládají vyčíslené hodnoty a proměnná count, která zde slouží jako index. Cyklus foreach celou sekvenci projde a pokud se proměnná count bude rovnat velikosti size, cyklus se ukončí. Pomocí klíčového slova yield se toto pole vrátí, když jsou jeho data potřeba. Tedy pokud CPU přistupuje zrovna k těmto datům, tak se vyčíslí. Tímto se zabrání zbytečnému zaplnění paměti RAM a nepotřebujeme paměť tak velkou, aby pojmula všechny kombinace, ale pouze její malou část. Poté už se jenom pole a proměnná count vynuluje, aby byly připraveny na další část. Díky podmínce v poslední části kódu máme zajištěno, že pokud nám nějaké prvky zbydou, budou také vráceny.

4.4.6 Načítání dat do pole využitím všech CPU vláken

Pomocí metody popsané v odstavci 4.4.5 si sekvenci rozdělíme a poté s daty pracujeme paralelně na všech CPU vláknech. V následujících dvou kódech si ukážeme, jak.

```
double maxCombsGPU = device.MaxMemoryAllocationSize / 60 / materialsCount;
int partsCount = (int)Math.Ceiling(combsCount / maxCombsGPU);
int threadCount = Environment.ProcessorCount, partGPU = combsCount /
    partsCount, partThread = partGPU / threadCount, c = 0;
IEnumerable<IEnumerable<IEnumerable<ChargeMaterial>>> combsInParts =
    combinations.Batch(partThread);
Task[] tasks = new Task[threadCount];
```

Výpis 4.10: Rozdělení na části

Nejprve si musíme ve Výpisu 4.10 zjistit důležité parametry, než budeme kombinace rozdělovat. V prvním řádku zjistíme, kolik kombinací se najednou vleze do paměti GPU. Proměnná device je OpenCL zařízení, se kterým pracujeme a pomocí parametru MaxMemoryAllocationSize zjistíme, jaká je nejvyšší možná velikost jednoho objektu v bytech, který na GPU pošleme. Toto číslo podělíme 60, protože jeden objekt struktury Material zabere 60 bytů paměti a počtem materiálů (materialsCount).

Ve druhém řádku zjistíme počet částí, po kterých budeme kombinace posílat na GPU. Proměnná combsCount je počet všech kombinací.

Do proměnné threadCount si uložíme počet vláken CPU, tedy počet logických procesorů, který zjistíme pomocí třídy Environment a její proměnné ProcessorCount. V tomto řádku dále zjistíme, kolik kombinací připadá na jednu část (partGPU), kolik kombinací připadá na jedno CPU vlákno (partThread) a nakonec si definujeme pomocnou proměnnou c, která nám v dalším kódu pomůže zjistit, zda je část dokončena.

Zavoláním metody Batch (Výpis 4.9) na všechny vygenerované kombinace (combinations) si kombinace rozdělíme na potřebný počet částí. Velikost jedné části je proměnná partThread. V posledním řádku si definujeme pole Task o velikosti počtu vláken, které fungují na bázi třídy Thread.

```
foreach(IEnumerable<IEnumerable<ChargeMaterial> sequence in combsInParts)
{
   int i = partThread * c * materialsCount;
   tasks[c] = Task.Run(() =>
   {
      foreach (IEnumerable<ChargeMaterial> item2 in sequence)
      {
         foreach (ChargeMaterial item in item2)
         {
            Material a = new Material();
            a.InflowAl2O3 = item.InflowAl2O3;
            ...
            materials[i] = a;
            i++;
         }
      }
   });
   if (c == threadCount - 1)
   {
      Task.WaitAll(tasks);
      Calculate(materials, resultsArr, material7, results);
      c = 0;
   }
   else
      c++;
}
```

Výpis 4.11: Načítání dat do pole pomocí CPU vláken

Ve Výpisu 4.11 můžeme vidět, co se na každém vláknu odehrává. Nejprve se vypočítá číslo i, které zde slouží pro to, aby každé vlákno zapisovalo do jiné části pole. Následně se vytváří úkoly, kde každý z nich čte jinou sekvenci a přepisuje data ze třídy ChargeMaterial do struktury Material. Využívá se struktura, protože GPU s třídou pracovat neumí a navíc na výpočet nepořebujeme všechny proměnné třídy ChargeMaterial, ale jen část, takže by to byla zbytečná data navíc.

Ve spodní části kódu máme podmínku, která nám určuje, kolikáté vlákno právě definujeme. Pokud c, které se na konci každé iterace inkrementuje, jak lze vidět na posledním řádku, má hodnotu počtu vláken - 1, tak se na všechna vlákna počká pomocí funkce Task.WaitAll a zavolá se funkce Calculate, která se postará o výpočet na GPU. Na všechna vlákna je třeba počkat, protože pak by běžela na pozadí, zatímco by se funkce Calculate spustila, a je právě potřeba, aby všechny materiály byly do pole uloženy úspěšně. Parametry funkce Calculate jsou po řadě: materiály, pole výsledků, které nám vrátí grafická karta, pole material7, které je vysvětleno v kapitole 4.4.8 a seznam results, do kterého se uloží jen výsledky, které splnily podmínku. Nakonec se vynuluje proměnná c, aby byla připravena na další část.

4.4.7 Porovnání výpočtů

Tabulka 4.4: Porovnání časů výpočtů vsázky

Porovnání výpočtů GPU vs CPU v sekundách							
	1,7 mil.		**14,7 mil.**		**147 mil.**		**Rozdíl**
	CPU	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**	
PC1	4,7	1,4	40,1	10,5	420,4	104,7	3,35 - 4,01x
PC4	12,9	3,9	123,4	34,4	-	330	3,3 - 3,58x

Jak můžeme vidět v Tabulce 4.4, zde pracujeme s mnohem většími objekty, než byly objekty popsané v kapitole 4.3 a výpočty jsou náročnější. Nejdéle trvá přesun dat z CPU na GPU. Rozdíly tedy nejsou desetinásobné, ale jsou i tak velmi velké. Pro vyhodnocení jsem neměl k dispozici všech šest PC. Specifikace jsou stejné jako v Tabulce 4.1. Porovnáváme zde tedy nejslabší a nejvýkonnější PC z těch původních. Opět lze dobře vidět, že čím více máme kombinací, tím více se výpočet na GPU vyplatí.

U PC1 je při 1,7 mil kombinací GPU 3,35x rychlejší, při 14,7 mil. už je to 3,82x a při 147 mil. je to 4,01x. PC2 je na tom velmi podobně. 1,7 mil. kombinací je na GPU 3,3x rychlejší a 14,7 mil. 3,58x rychlejší. 147 mil. na CPU počítat nemůžeme, jelikož by čas přesáhl přibližně 20 minut. Větší počet kombinací testovat nemůžeme, protože klient na odpověď od serveru čeká 10 minut, a pokud nedostane do tohoto času odpověď, odpojí se.

4.4.8 Fáze zlepšování a zrychlení algoritmu na GPU

Algoritmus pro technologickou a finanční kalkulaci vsázky pro výrobu aglomerátu a surového železa se postupem času a vývoje zrychloval. Rozhodl jsem se, že postupné úpravy, změny a vylepšení rozdělím do jednotlivých fází, kde změny popíšu. U jednotlivých fází jsou i časy, jak dlouho trvaly při výpočtu 1,7 mil. kombinací na PC1.

První funkční výpočet vsázky, který vrací správný počet výsledků, trval 32 sekund. Všechny operace a přesuny dat se provádí na jednom CPU vlákně, jen samotný výpočet se provádí na grafické kartě.

Načítání kombinací ze sekvence do pole pomocí Parallel.For, kde je využito všech CPU vláken. Celková doba výpočtu činila 23 sekund.

Výpočty proměnných cokeQuantity, limeQuantity a matQuantity jsou přesunuty na grafickou kartu, kde pro ně byla vytvořena funkce, snížily čas výpočtu o 6 sekund na 17 sekund.

V další fázi je kontrola výsledků přímo na GPU a vrací se pouze ty, které vyhovují. GPU postupně počítá jeden parametr výsledku za druhým, a pokud zjistí, že jeden ze zatím vypočtených parametrů nevyhovuje uživatelem zadaným hodnotám, výpočet ukončí. Celkový čas provedení byl 9 sekund.

U všech proměnných je změněn datový typ z double na float. Tímto se zabraná paměť zmenšila na polovinu a zároveň se samotný výpočet zrychlil. Rozdíl ve výsledcích je velmi nepatrný. Čas trvání činil 5,1 sekund.

Materiálů je jen zadaný počet, např. 7, jejich hodnoty se tedy každou kombinaci opakují, pouze parametr Quantity se mění. Přidáním struktury Material7, kde těchto 7 materiálů uložíme, ušetříme paměti a dosáhneme zrychlení. Nemusíme tak tyto hodnoty ukládat tolikrát, kolik máme kombinací. Čas klesl na 3,5 sekund.

Výpočet vnosů (Inflows) probíhá na GPU, což způsobí nárůst potřebné paměti, ale znatelné zrychlení. Čas výpočtu byl 2,4 sekund.

Využití metody Batch na rozdělení podle počtu CPU vláken, kde každé vlákno zpracovává svou část. Rozdíl mezi druhou a touto fází je v tom, že Parallel.For automaticky pracuje s vlákny a vytvoří jich určitý počet, zatímco u tohoto bodu s pomocí třídy Task si vlákna vytvořím sám, tedy si můžu určit, jaký bude jejich počet atd. Konečný čas algoritmu se zastavil na 1,4 sekundách.

4.5 Návrh ideálního PC

Poslední úkol na praxi spočíval v navržení ideálního PC pro tyto výpočty. Výpočet pomocí grafické karty těží hlavně z toho, že její velký počet jader počítá paralelně. Abychom využili této výhody co nejvíce, je potřeba co největší paměť a to i především z toho důvodu, že přesun dat z CPU na GPU zabírá dost času, např. pokud se podíváme na Obrázek 4.2, kde v případě PC1 z celkového času 1,4s samotný výpočet trvá pouze 216ms, zbylou 1 sekundu trvá přesun dat. V této

fázi hodně záleží na frekvenci procesoru a počtu jeho vláken, a proto je potřeba co největší paměť, aby bylo přesunů dat co nejméně. To platí jak pro paměť RAM, tak paměť GPU.

Pro nejlepší výkon jsou důležité tyto tři komponenty:

- Grafická karta
- Procesor
- Paměť RAM

U grafické karty máme hned několik důležitých parametrů, které velmi ovlivní rychlost výpočtů. Nejdůležitější je její paměť, aby se do ní dalo uložit co nejvíce kombinací, počet a frekvenci jader, které ovlivní celkovou rychlost. Pro nás nejlepší možností je nová grafická karta od společnosti NVIDIA. Jedná se o model Geforce RTX 3090, který je nyní nejvýkonnější grafickou kartou na trhu. Disponuje 24 GB GDDR6X paměti a počtem 10 496 CUDA jader, které se přetaktují až na frekvenci 1,7 GHz. Pro dosažení nejlepšího výkonu by se tato GPU v této PC sestavě nacházela hned 4x. GPU podporuje PCIe 4.0, díky které má mnohonásobně vyšší propustnost, která zaručuje, že se data z CPU na GPU přesunou rychleji. Jedna karta při plném výkonu vyžaduje výkon 350 wattů. [16]

Pro odeslání dat na GPU je zapotřebí výkonný procesor. Nejdůležitějšími parametry je počet jader a vláken, které se dobře využijí pro paralelní čtení ze sekvence a zápis do polí. Velmi důležitou roli zde hraje frekvence jader, která nám udává, jak rychle každé jádro tyto příkazy zpracuje. Abychom mohli využít čtyři grafické karty, kde každá vyžaduje 16 PCIe linek čtvrté generace, je zapotřebí procesor, který disponuje 64 těmito linkami. Ryzen Threadripper 3970X od společnosti AMD všem těmto parametrům vyhovuje. K dispozici má 32 jader a 64 vláken, které dosáhnou až na frekvenci 4,5 GHz. Obsahuje dohromady 88 PCIe linek čtvrté generace. [17]

Paměť RAM nám slouží k uložení dat, než je odešleme na GPU. Výše zmíněný procesor nám také nabídne quad channel, tzn. rychlejší přesun dat do paměti. Dále je důležitá její kapacita, která musí být minimálně stejně velká jako paměť GPU, jelikož se zde jedná o stejně velké objekty. Dalším důležitým parametrem je její frekvence. Všechny čtyři GPU mají dohromady paměť 96 GB, tedy ideální volbou by byla paměť o velikosti 128 GB a frekvenci 3,2 GHz. Všem parametrům vyhovuje např. Kingston HyperX 128GB KIT DDR4 3200MHz CL16 FURY Black series, která je poskládána ze čtyř pamětí po 32 GB. [18]

Ostatní komponenty musí také splňovat určitá kritéria. Základní deska musí obsahovat patici sTRX4, kterou vyžaduje výše zmíněný procesor. Dále musí podporovat paměť RAM DDR4 o velikosti 128GB s podporou quad channel. Musí obsahovat 4 PCIe 4.0 x16 sloty pro výše zmíněné grafické karty. Zdroj by měl mít kapacitu 2 000 wattů, aby zvládl napájet všechny čtyři grafické karty a procesor.

Kapitola 5

Znalosti a dovednosti získané během studia a uplatněné v průběhu odborné praxe

V této bakalářské práci jsem nejvíce čerpal z předmětu Programovací jazyky II, kde jsem se učil programovací jazyk C#, který je v projektech nejvíce využitý. Dále předměty Programování I/II a Algoritmy I/II, kde jsem se naučil základy programovacího jazyka C++, který je dost podobný jazyku C99 a který jsem využil při programování výpočtů na grafické kartě. K výpočtům mi také pomohl předmět Architektury počítačů a paralelních systémů, kde jsem se naučil, jak pracovat s vlákny a také jak funguje technologie CUDA, která je velmi podobná OpenCL. V předmětech Úvod do databázových systémů a Databázové a informační systémy jsem se naučil, jak pracovat s databází a jak tyto technologie využít na tvorbu informačních systémů. Předměty Architektura technologie .NET a Vývoj informačních systémů pomohly k rozšíření znalostí jazyka C#. V závěru mi pomohl předmět Elektronické publikování, který se zabývá technologií Latex, kterou jsem využil pro psaní této bakalářské práce.

Kapitola 6

Závěr

V průběhu odborné praxe ve firmě Liberty Ostrava a.s. jsem všechny zadané úkoly i úkoly dílčí splnil. Ve firmě budu pracovat i v příštím období a rád bych se i nadále věnoval optimalizacím pro výpočet vsázky.

Vedení firmy bylo velmi vstřícné i k mému časovému rozvrhu, neboť prostor při studiu posledního ročníku bakalářského studia byl omezen a nutil mě tedy daný čas využít smysluplně. Nejvíce si cením získaných znalostí a dovedností při výpočtech na grafických kartách, kdy jsem si sám ověřil, jak velkého zrychlení je možné dosáhnout.

V rámci pracovního působení ve firmě jsem se naučil plnit zadané úkoly rychleji a zároveň i efektivněji.

Podle mého názoru je bakalářská práce formou praxe vhodnější než zadávaná témata. Ve firmě se pracuje v týmu na reálných projektech, přičemž jsem si veškeré teoretické znalosti získané během studia výrazně obohatil o zkušenosti a dovednosti praktické.

Reference

1. *Domů - LIBERTY Steel Czech Republic* [online]. Ostrava: LIBERTY Steel Group, 2021 [cit. 2021-03-18]. Dostupné z: https://libertysteelgroup.com/cz/.
2. *Historie společnosti - LIBERTY Steel Czech Republic* [online]. Ostrava: LIBERTY Steel Group, 2021 [cit. 2021-03-18]. Dostupné z: https://libertysteelgroup.com/cz/o-spolecnosti/historie-spolecnosti/.
3. *Lakshmi Mittal - Chairman and CEO of Mittal Steel* [online]. Kochi: Suni Systems (P), 2021 [cit. 2021-03-18]. Dostupné z: https://www.webindia123.com/personal/abroad/mittal.htm.
4. *Lekce 1 - Úvod do C# a .NET frameworku* [online]. Praha: Itnetwork.cz, 2021 [cit. 2021-03-18]. Dostupné z: https://www.itnetwork.cz/csharp/zaklady/c-sharp-tutorial-uvod-do-jazyka-a-dot-net-framework.
5. *Introduction - C# language specification | Microsoft Docs* [online]. Redmond: Microsoft, 2021 [cit. 2021-03-18]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction.
6. *Introduction to C# - GeeksforGeeks* [online]. Noida: GeeksforGeeks, 2021 [cit. 2021-03-18]. Dostupné z: https://www.geeksforgeeks.org/introduction-to-c-sharp/.
7. *Language-Integrated Query (LINQ) (C#) | Microsoft Docs* [online]. Redmond: Microsoft, 2021 [cit. 2021-03-18]. Dostupné z: GeeksforGeeks|Acomputerscienceportalforgeeks.
8. *SQLCourse - Lesson 1: What is SQL?* [Online]. Nashville: TechnologyAdvice, 2021 [cit. 2021-03-18]. Dostupné z: http://www.sqlcourse.com/intro.html.
9. *Part 1: RabbitMQ for beginners - What is RabbitMQ? - CloudAMQP* [online]. Švédsko: CloudAMQP, 2021 [cit. 2021-03-18]. Dostupné z: https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html.
10. *OpenCL Overview - The Khronos Group Inc* [online]. Beaverton: The Khronos Group, 2021 [cit. 2021-03-17]. Dostupné z: https://www.khronos.org/opencl/.

11. *OpenCL Basics* [online]. Jülich: Forschungszentrum Jülich, c2021 [cit. 2021-03-18]. Dostupné z: https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/opencl/opencl-03-basics.pdf?__blob=publicationFile.

12. *Pros and Cons of OpenCL for FFT* [online]. Meetup, 2021 [cit. 2021-03-18]. Dostupné z: http://files.meetup.com/1774957/pros-and-cons-of-opencl-updated.pdf.

13. *OpenCL Memory model [1] | Download Scientific Diagram* [online]. Berlín: ResearchGate, 2021 [cit. 2021-03-26]. Dostupné z: https://www.researchgate.net/figure/OpenCL-Memory-model-1_fig1_275973210.

14. *How to Use Your GPU in .NET - CodeProject* [online]. CodeProject, 2021 [cit. 2021-04-13]. Dostupné z: https://www.codeproject.com/Articles/1116907/How-to-Use-Your-GPU-in-NET.

15. *MoreLINQ/Batch.cs at master · morelinq/MoreLINQ · GitHub* [online]. San Francisco: GitHub, 2021 [cit. 2021-04-13]. Dostupné z: https://github.com/morelinq/MoreLINQ/blob/master/MoreLinq/Batch.cs.

16. *GeForce RTX 3090 Graphics Card | NVIDIA* [online]. Santa Clara: NVIDIA Corporation, 2021 [cit. 2021-03-29]. Dostupné z: https://www.nvidia.com/en-gb/geforce/graphics-cards/30-series/rtx-3090/.

17. *3rd Gen Ryzen™ Threadripper™ 3970X | Desktop Processor | AMD* [online]. Sunnyvale: Advanced Micro Devices, 2021 [cit. 2021-03-29]. Dostupné z: https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3970x.

18. *FURY DDR4 RGB Memory – 8GB-128GB/2400MHz-3733MHz | HyperX* [online]. Fountain Valley: Kingston Technology Corporation, 2021 [cit. 2021-03-29]. Dostupné z: https://www.hyperxgaming.com/czech/us/memory/fury-ddr4-rgb.

Příloha A

Výpočet vsázky na grafické kartě

```
constant float CMin = 4.5, CMax = 4.5, SiMin = 0.75, SiMax = 0.75, YieldMin =
    10, YieldMax = 20, ChargeRichnessMin = 50, ChargeRichnessMax = 60,
    BasicityMin = 1.05, BasicityMax = 1.15, MnMin = 0.3, MnMax = 0.5, MgOMin =
    9, MgOMax = 11, SlaQtyMin = 440, SlaQtyMax = 470,
    CokeSpecificConsumptionMin = 450, CokeSpecificConsumptionMax = 520,
    constantsFe = 0.99, constantsMn = 0.6334, constantsP = 0.95, constantsS =
    0.05, constantsZn = 0.3982, constantsTi = 0.0255, constantsCr = 0.64,
    burdenRequestCokeAsh = 8.6, burdenRequestBasicity = 1.1, burdenRequestSi =
    0.75, burdenRequestC = 4.5;
constant int materialsCount = 7;

typedef struct
{
int Quantity;
float InflowAl2O3, InflowMgO, InflowSiO2, InflowCaO, PriceForQuantity,
    InflowFe, InflowMn, InflowP, InflowZn, InflowCr, InflowS, InflowTiO2,
    InflowK2O, InflowNa2O;
} Material;

typedef struct
{
int ChargeMatGroupID;
float Al2O3, CaO, Fe, MgO, SiO2;
} Material7;

typedef struct
```

```
{
float InflowFe, InflowMn, InflowP, InflowZn, InflowCr, InflowS, InflowTi,
    InflowAlcalic, InflowSiO2, InflowAl2O3, InflowCaO, InflowMgO,
    HELP_VARIABLE, InflowFeSiO2;
} Inflows;

typedef struct
{
float resultSlagQty, resultTheorFe, resultTheorMn, resultYield, resultTheorSi,
    resultTheorP, resultTheorS, resultTheorZn, resultTheorTi, resultTheorCr,
    resultTheorC, resultTheorMgO, resultTotalOre, resultHotMetalQuantity,
    resultSpecificPrice, resultMaterialQuantity,
    resultChargeRichness,resultInflowAlcalic, resultInflowCr, resultInflowFe,
    resultInflowMn, resultInflowP, resultInflowS, resultInflowTi,
    resultInflowZn, resultCokeSpecificConsumption,
    resultCokeSpecificConsumptionCarbonated, resultCokeLoadCapacity,
    resultTotalLimestone, resultBasicity;
} Result;

float GetQuantityInCharge(Material* materials, int groupID, global Material7*
    materials7)
{
   double result = 0;
   for (int i = 0; i < materialsCount; i++)
   {
      if (groupID == 0)
      {
         result += materials[i].Quantity;
      }
      else if (materials7[i].ChargeMatGroupID == groupID)
      {
         result += materials[i].Quantity;
      }
   }
return result;
}

Inflows SetInflows(Material * materialsInCharge)
```

```
{
   Inflows inflows;
   for (int i = 0; i < materialsCount; i++)
   {
      inflows.InflowFe = inflows.InflowFe + materialsInCharge[i].InflowFe;
      inflows.InflowMn = inflows.InflowMn + materialsInCharge[i].InflowMn;
      inflows.InflowP = inflows.InflowP + materialsInCharge[i].InflowP;
      inflows.InflowZn = inflows.InflowZn + materialsInCharge[i].InflowZn;
      inflows.InflowCr = inflows.InflowCr + materialsInCharge[i].InflowCr;
      inflows.InflowS = inflows.InflowS + materialsInCharge[i].InflowS;
      inflows.InflowTi = inflows.InflowTi + materialsInCharge[i].InflowTiO2;
      inflows.InflowAlcalic = inflows.InflowAlcalic +
         materialsInCharge[i].InflowK2O + materialsInCharge[i].InflowNa2O;
      inflows.InflowSiO2 = inflows.InflowSiO2 + materialsInCharge[i].InflowSiO2;
      inflows.InflowAl2O3 = inflows.InflowAl2O3 +
         materialsInCharge[i].InflowAl2O3;
      inflows.InflowCaO = inflows.InflowCaO + materialsInCharge[i].InflowCaO;
      inflows.InflowMgO = inflows.InflowMgO + materialsInCharge[i].InflowMgO;
   }
   inflows.HELP_VARIABLE = (inflows.InflowFe * constantsFe + inflows.InflowMn
      * constantsMn + inflows.InflowP * constantsP + inflows.InflowS *
      constantsS + inflows.InflowZn * constantsZn + inflows.InflowTi *
      constantsTi + inflows.InflowCr * constantsCr) / 1000 * 100 / (100 -
      (burdenRequestSi + burdenRequestC));
   inflows.InflowFeSiO2 = inflows.HELP_VARIABLE * ((burdenRequestSi / 100) *
      (60.0 / 28.0)) * 1000;
return inflows;
}

kernel void ComputeGPU(global Material* materialsInCharge, global Material7*
    materials7, global Result* result)
{
   int limeGroupID = 13, fuelGroupID = 20;
   Material7 m13[materialsCount];
   Material m7[materialsCount];
   int indexes[materialsCount];
   int index = get_global_id(0);
   int j = get_global_id(0);
```

```
int num = 0;

float totalAcid, totalBasic, xv, yv, materialWithoutFuelQuantity;
float C, Si, Yield, ChargeRichness, Basicity, Mn, MgO, SlaQty,
    CokeSpecificConsumption;

for (int i = 0; i < materialsCount; i++)
{
   m7[i] = materialsInCharge[i + index * materialsCount];
}
Inflows inflows = SetInflows(m7);

float cokeQuantity = GetQuantityInCharge(m7, fuelGroupID, materials7);
float limestoneQuantity = GetQuantityInCharge(m7, limeGroupID, materials7);
float materialQuantity = GetQuantityInCharge(m7, 0, materials7);
materialWithoutFuelQuantity = materialQuantity - cokeQuantity;

Si = burdenRequestSi;
C = burdenRequestC;
Yield = inflows.InflowFe / 1000;

if (inflows.InflowFe == 0)
   CokeSpecificConsumption = 0;
else
   CokeSpecificConsumption = 100000 * cokeQuantity / inflows.InflowFe;

if ((inflows.InflowSiO2 - inflows.InflowFeSiO2 + inflows.InflowAl2O3 +
    inflows.InflowCaO + inflows.InflowMgO) == 0)
   MgO = 0;
else
   MgO = (100 * inflows.InflowMgO) / (inflows.InflowSiO2 -
       inflows.InflowFeSiO2 + inflows.InflowAl2O3 + inflows.InflowCaO +
       inflows.InflowMgO);

if ((inflows.InflowSiO2 - inflows.InflowFeSiO2 + inflows.InflowAl2O3) == 0)
   Basicity = 0;
else
```

```
   Basicity = (inflows.InflowCaO + inflows.InflowMgO) / (inflows.InflowSiO2
      - inflows.InflowFeSiO2 + inflows.InflowAl2O3);

if (materialWithoutFuelQuantity == 0)
   ChargeRichness = 0;
else
   ChargeRichness = 100 * ((inflows.InflowFe / materialWithoutFuelQuantity)
      + (inflows.InflowMn / materialWithoutFuelQuantity)) / (100 +
      (limestoneQuantity / materialWithoutFuelQuantity) * 100);

if (inflows.HELP_VARIABLE == 0)
   SlaQty = 0;
else
   SlaQty = (inflows.InflowSiO2 - inflows.InflowFeSiO2 + inflows.InflowAl2O3
      + inflows.InflowCaO + inflows.InflowMgO) / inflows.HELP_VARIABLE;

if (inflows.HELP_VARIABLE == 0)
   Mn = 0;
else
   Mn = (inflows.InflowMn * constantsMn) / (inflows.HELP_VARIABLE * 10);


if (C >= CMin && C <= CMax && Si>= SiMin && Si <= SiMax && Yield >=
    YieldMin && Yield <= YieldMax && ChargeRichness >= ChargeRichnessMin &&
    ChargeRichness <= ChargeRichnessMax)
{
   if (Basicity >= BasicityMin && Basicity <= BasicityMax && Mn >= MnMin &&
      Mn <= MnMax && MgO >= MgOMin && MgO <= MgOMax && SlaQty >= SlaQtyMin
      && SlaQty <= SlaQtyMax && CokeSpecificConsumption >=
      CokeSpecificConsumptionMin && CokeSpecificConsumption <=
      CokeSpecificConsumptionMax)
   {
      int l = 0, inc = 0;

      for (int i = 0 + index * materialsCount; i < materialsCount + index *
         materialsCount; i++)
      {
         if (materials7[inc].ChargeMatGroupID == limeGroupID)
```

```
{
m13[l] = materials7[inc];
indexes[l] = i;
l++;
num++;
}

totalAcid = totalAcid + materialsInCharge[i].InflowSiO2 +
    materialsInCharge[i].InflowAl2O3;
totalBasic = totalBasic + materialsInCharge[i].InflowCaO +
    materialsInCharge[i].InflowMgO;
inc++;
}

if (limestoneQuantity == 0)
{
   for (int i = 0; i < l; i++)
   {
      xv = xv + m13[i].CaO + m13[i].MgO;
      yv = yv + m13[i].SiO2 + m13[i].Al2O3;
   }
   xv = xv / (l);
   yv = yv / (l) * burdenRequestBasicity;
}
else
{
   for (int i = 0; i < l; i++)
   {
      xv = xv + (m13[i].CaO + m13[i].MgO) *
          materialsInCharge[indexes[i]].Quantity;
      yv = yv + (m13[i].SiO2 + m13[i].Al2O3) *
          materialsInCharge[indexes[i]].Quantity;

   }
   xv = xv / limestoneQuantity;
   yv = yv / limestoneQuantity * burdenRequestBasicity;
}
```

```
if (inflows.InflowFe == 0)
   result[j].resultCokeSpecificConsumptionCarbonated = 0;
else
   result[j].resultCokeSpecificConsumptionCarbonated = 100000 *
       (cokeQuantity - (cokeQuantity * burdenRequestCokeAsh / 100)) /
       inflows.InflowFe;

if (cokeQuantity == 0)
   result[j].resultCokeLoadCapacity = 0;
else
   result[j].resultCokeLoadCapacity = (materialQuantity -
       cokeQuantity) / cokeQuantity;

if (yv - xv == 0)
   result[j].resultTotalLimestone = 0;
else
   result[j].resultTotalLimestone = (burdenRequestBasicity * totalAcid
       - totalBasic) / (yv - xv);

result[j].resultInflowAlcalic = inflows.InflowAlcalic;
result[j].resultInflowCr = inflows.InflowCr;
result[j].resultInflowFe = inflows.InflowFe;
result[j].resultInflowMn = inflows.InflowMn;
result[j].resultInflowP = inflows.InflowP;
result[j].resultInflowS = inflows.InflowS;
result[j].resultInflowTi = inflows.InflowTi;
result[j].resultInflowZn = inflows.InflowZn;

if (inflows.HELP_VARIABLE == 0)
{
   result[j].resultTheorFe = 0;
   result[j].resultTheorP = 0;
   result[j].resultTheorS = 0;
   result[j].resultTheorZn = 0;
   result[j].resultTheorTi = 0;
   result[j].resultTheorCr = 0;
}
else
```

```
{
   result[j].resultTheorFe = (inflows.InflowFe * constantsFe) /
       (inflows.HELP_VARIABLE * 10);
   result[j].resultTheorP = (inflows.InflowP * constantsP) /
       (inflows.HELP_VARIABLE * 10);
   result[j].resultTheorS = (inflows.InflowS * constantsS) /
       (inflows.HELP_VARIABLE * 10);
   result[j].resultTheorZn = (inflows.InflowZn * constantsZn) /
       (inflows.HELP_VARIABLE * 10);
   result[j].resultTheorTi = (inflows.InflowTi * constantsTi) /
       (inflows.HELP_VARIABLE * 10);
   result[j].resultTheorCr = (inflows.InflowCr * constantsCr) /
       (inflows.HELP_VARIABLE * 10);
}

result[j].resultTotalOre = materialWithoutFuelQuantity;

inc = 0;
for (int i = 0 + index * materialsCount; i < materialsCount + index *
    materialsCount; i++)
{
   if (materials7[inc].Fe >= 0.7)
     result[j].resultHotMetalQuantity =
         result[j].resultHotMetalQuantity + materials7[inc].Fe *
         materialsInCharge[i].Quantity * 100 / 1000;

   result[j].resultSpecificPrice = result[j].resultSpecificPrice +
       materialsInCharge[i].PriceForQuantity;
   result[j].resultMaterialQuantity = result[j].resultMaterialQuantity
       + materialsInCharge[i].Quantity;
   inc++;
}
result[j].resultSpecificPrice = result[j].resultSpecificPrice /
    result[j].resultHotMetalQuantity;
result[j].resultTheorC = C;
result[j].resultTheorSi = Si;
result[j].resultYield = Yield;
result[j].resultChargeRichness = ChargeRichness;
```

```
            result[j].resultBasicity = Basicity;
            result[j].resultTheorMn = Mn;
            result[j].resultTheorMgO = MgO;
            result[j].resultSlagQty = SlaQty;
            result[j].resultCokeSpecificConsumption = CokeSpecificConsumption;
        }
    }
}
```

Výpis A.1: Výpočet vsázky na GPU